Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 271–280

Procedia Computer Science

International Conference on Computational Science, ICCS 2011

OpenMP parallelization of the SCIARA Cellular Automata lava flow model: performance analysis on shared-memory computers

Marco Oliverio[a], William Spataro[a]*, Donato D'Ambrosio[a], Rocco Rongo[b], Giuseppe Spingola[a], Giuseppe A. Trunfio[c]

[a]*Department of Mathematics, University of Calabria, Rende 87036, Italy*
[b]*Department of Earth Sciences, University of Calabria, Rende 87036, Italy*
[c]*DADU, University of Sassari, Alghero 07041, Italy*

Abstract

Parallel Computing represents a valid solution for reducing execution times in simulations of complex geological processes, such as lava flows, debris flows and, in general, of fluid-dynamic processes. In these cases, Cellular Automata (CA) models have proved to be effective when the behavior of the system to be modeled can be described in terms of local interactions among its constituent parts. Cellular Automata are parallel computing models, discrete in space and time; space is generally subdivided into cells of uniform size and the overall dynamics of the system emerges as the result of the simultaneous application, at discrete time steps, of proper local rules of evolution to each one of them. Due to their intrinsic parallelism, CA models are attractive since they are suitable to be effectively and naturally implemented on parallel computers achieving also high performance. In the recent past, CA models were efficiently executed on distributed memory architectures, such as Beowulf clusters and many-node Supercomputers, while fewer implementations are found regarding shared-memory computers, such as in multi-core machines. This paper shows performance results of the parallelization of a well-known CA model for simulating lava flows – the SCIARA model – in a shared memory environment, by means of OpenMP, an Application Programming Interface which supports multi-platform shared-memory parallel programming.

Keywords: Parallelization; Shared Memory paradigm, OpenMP; Cellular Automata; Lava flow modelling

1. Introduction

Parallel Computing [1] refers to a type of computation where different calculations are carried out simultaneously, operating on the principle that large problems can be split into smaller ones, which are then solved concurrently. The fast development of High Performance Computing (HPC) has in fact allowed the use of numerical simulations as a tool for solving complex equation systems, in which researchers can study the modelling of real complex phenomena, such as lava flows, fire spreading and traffic simulation. As a consequence, the developer must implement proper optimization strategies and, if possible, parallelize the program. The type of parallelization

* Corresponding author. Tel.: + 39-0984-49-4875; fax: +39-0984-49-3570.
E-mail address: spataro@unical.it.

Selection and/or peer-review under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.029

needed in this latter phase depends on the kind of available parallel architecture. For instance, in case of a distributed memory machine, this can be accomplished by means of MPI – Message Passing Interface [2]. At the contrary, if a multicore (and shared memory) architecture machine is available, a shared-memory or multithread implementation can result in a better and more efficient solution. In this latter specific field of parallel systems, OpenMP [3] defines specifications for parallelizing programs in a shared memory environment. In particular, OpenMP represents the most widely adopted solution where compiler directives and environment variables specify shared-memory parallelism in a quasi-transparent way.

The study of geological processes is of increasing interest for the Scientific Community both for theoretical aspects and possible practical applications related to risk assessment and mitigation. Through the simulations of reliable models, risks associated to such processes can be evaluated and possible countermeasures can be conjectured and tested. Among different methodologies used for modeling these type of phenomena (and thus related to the Computational Science field) such as numerical analysis, high order difference approximations and finite differences, Cellular Automata (CA) [4] proved to be particularly suitable when the behavior of the system to be modeled can be described in terms of local interactions. Originally introduced by von Neumann in the 50s to study self-reproduction issues, CA are discrete computational models, widely utilized for modeling and simulating complex systems. Well known examples are Lattice Gas Automata and Lattice Boltzmann models [5] which are particularly suitable for modeling fluid dynamics at a microscopic level of description. However, many complex phenomena (e.g. landslides or lava flows) are difficult to be modeled at such scale, as they generally evolve on large areas, thus needing a macroscopic level of description. Moreover, they may be also difficult to be modeled through standard approaches, such as differential equations (cf. [6] for the case of lava flows), and Macroscopic Cellular Automata (MCA) [7] can represent a valid alternative. As the name suggests, the main features of the phenomena of interest can be directly described at a macroscopic level (e.g. average amount of lava, temperature, etc), thus disregarding microscopic aspects. Well known examples of MCA applications include the simulation of lava [8], debris flows [9], pyroclastic flows [10], density currents [11], water flux in unsaturated soils [12], Artificial Life systems [13] and forest fires [14], besides many others. Refer to [7] for a more detailed description, besides theoretical aspects on MCA and, for instance, to the ACRI conference (e.g. [15]) series for up-to-date general CA applications interested by the International Scientific Community.

Several successful attempts have been carried out regarding solutions for automatically parallelizing MCA simulation models. Among others, CAMELot [16] and libAuToti [17] represent valid solutions for implementing MCA models on distributed memory machines, such as Beowulf clusters and Supercomputers, where good performances where achieved. Notwithstanding, with the emergence of multi-processor computers, multithreaded programming is quickly gaining popularity and the adoption of shared-memory programming implementations could be more appropriate for these architectures. Nonetheless, few valid OpenMP parallelizations have been implemented for CA-like models, such as for fire spread simulation [18] or as in [19] where interesting results were obtained for the parallelization of Lattice Boltzmann models.

This paper presents first results related to the parallelization, in shared memory environments, of a well-known, reliable and efficient model widely adopted for lava flow risk assessment, the SCIARA MCA model. First tests have shown the validity of this kind of approach that can be considered also effective for hybrid parallel solutions (i.e. jointly usage of MPI and OpenMP). In the following, after a brief description of the latest version of the SCIARA MCA model for lava flows, a quick overview of the shared memory paradigm and of the OpenMP library are presented. Subsequently, the specific implementation and performance analysis referred to simulations of three real lava flow events occurred on Mt. Etna (Italy) are shown, while conclusions are reported at the end of the paper.

2. The SCIARA Cellular Automata lava flow computational model

In general, a computational model for simulating lava flows should be well calibrated and validated against several test cases in order to be fruitfully and reliably applied for its application for land use planning and civil defense purposes. Our research group is experienced in this field since 1982, when a first computational model of basaltic lava flows was proposed [20]. In recent years, we enriched the SCIARA family of lava flows simulation models, based on the Cellular Automata (CA) computational paradigm and, specifically, on the Macroscopic Cellular Automata (MCA) [7] approach for the modeling of spatially extended dynamical systems, by proposing improved releases and applying them to the simulation of diverse Etnean cases of study.

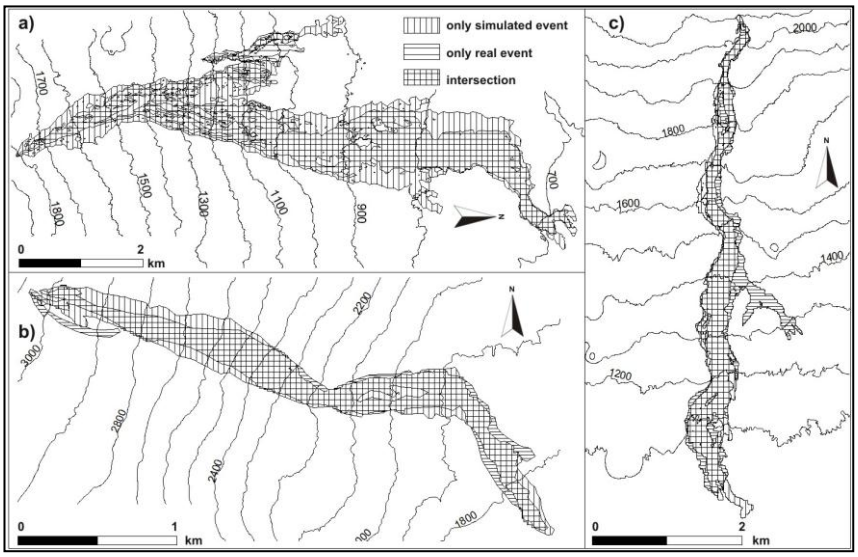

Fig. 1. Simulations of the 1981 (a), 2006 (b) and 2001 (c) Etnean lava flows by the CA model SCIARA. The key legend shows comparison between the simulated and real events.

Cellular Automata models are intrinsically parallel, so their implementation on parallel computers is straightforward and the simulation duration can be reduced almost proportionally to the number of available processors [21]. Regarding this work, an improved version of the SCIARA Cellular Automata model for simulating lava flows was adopted therefore. In this new model, a Bingham-like rheology and, therefore, the concepts of critical height and viscosity are explicitly considered ([22, 23]), has been introduced as part of the Minimization Algorithm of the Differences [7], which is applied for computing lava outflows from the generic cell towards its neighbors.

In formal terms, SCIARA is defined as:

$$SCIARA = < R, L, X, Q, P, \tau, \gamma >$$

where:

- R is the set of square cells covering the bi-dimensional finite region where the phenomenon evolves;
- $L \in R$ specifies the lava source cells (i.e. craters);
- X = {(0, 0), (0, 1), (-1, 0), (1, 0), (0, -1), (-1, 1), (-1,-1), (1, -1), (1, 1)} identifies the pattern of cells (Moore neighbourhood) that influence the cell state change; in the following we will refer to cells by indexes 0 (for the central cell) through 8;
- $Q = Q_z \times Q_h \times Q_T \times Q_f^8$ is the finite set of states, considered as Cartesian product of "substates". Their meanings are: cell altitude a.s.l., cell lava thickness, cell lava temperature, and lava thickness outflows (from the central cell toward the eight adjacent cells), respectively;
- $P=\{w, t, T_{sol}, T_{vent}, r_{Tsol}, r_{Tvent}, hc_{Tsol}, hc_{Tvent}, \delta, \rho, \varepsilon, \sigma, c_v\}$ is the finite set of parameters (invariant in time and space) which affect the transition function (please refer to [24] for their specifications);
- $\tau : Q^9 \rightarrow Q$ is the cell deterministic transition function, applied to each cell at each time step, which describes the dynamics of lava flows, such as cooling, solidification and lava outflows from the central cell towards neighbouring ones;
- $\gamma : Q_h \times \mathrm{N} \rightarrow Q_h$ specifies the emitted lava thickness from the source cells at each step $k \in N$ (N is the set of natural numbers).

The hexagonal cellular space adopted in the previous releases (e.g., cf. [8]) of the model for mitigating the anisotropic flow direction problem has been replaced by a - Moore neighborhood - square one, nevertheless by producing an even better solution for the anisotropic effect thanks to a better management of flows towards neighbouring cells [24]. Furthermore, many improvements have been introduced concerning the important modeling aspect of lava cooling. The model has been tested with good results by considering real cases of study occurred at

Mt Etna (Italy) and ideal surfaces, such as an inclined plane and an octagonal-base pyramid, in order to evaluate the magnitude of the anisotropic effect. Fig. 1 reports the simulation with SCIARA of three important real lava flow events which occurred on the volcano - namely the 1981, the 2001 and 2006 events - demonstrating the reliability of the new model. As a matter of fact, notwithstanding a preliminary calibration, the model demonstrated to be more accurate than its predecessors, providing the best results ever obtained on the simulation of the considered real cases of study. Please refer to [24] for details on these experiments and for further details of the new SCIARA model.

3. Shared memory parallel architectures and OpenMP

In computational science, the principle aim of parallel computing is to reduce the processing time of particularly complex simulations, thanks to the use of more processing units that collaborate to simultaneously solve the problem. Among others, M.J. Flynn has provided a comprehensive classification of parallel architectures where the category of a specific parallel computer depends on its ability to manage in parallel the instruction and/or data stream [25]. Another classification of parallel computers may be based on the kind memory architecture, whether shared or distributed. In distributed memory architectures, each processor unit has access to a different memory. In this case, it is necessary to have some kind of communication between the various processing units. In shared-memory architectures, all processing units access the same memory, even simultaneously. Each unit has a global view of memory, i.e., the address space is the same in all processors. This means that the code is more simple to write, since the programmer does not need to worry about data transmission between the various processing elements. In these systems, the discrepancy between the processor speed and the speed of the memory has always been the main bottleneck for the amount of operations that a processor can perform in a given time, because the former must wait that memory, significantly slower, provides the processor with the data. Mainly for this reason, when writing parallel code, it is essential to limit the number of memory accesses for maintaining high performance.

While absent in distributed memory architectures, where each processing element has its own local memory for data storage, *data coherence* represents one of the most important problems in shared memory architectures. In order to keep the consistency of memory, a parallel architecture has to necessarily transfer data between the various levels of cache and main memory. A processing unit may contain a value in its cache which is invalid, since updated in the main memory from another unit. To ensure that memory is always consistent, some invalidation and synchronization mechanism between the different levels of memory must be implemented. This task is seldom assigned to the programmer, since this would mean writing very complicated code, difficult to be read and with a high probability of error, and thus is delegated to the computer operating system or directly to hardware (such as in cache-coherent systems).

Another important issue concerning shared-memory architectures occurs when two or more units try to write simultaneously to the same memory location, determining what is called a *race condition*. The problem consists in the fact that the write operation is not an atomic one, i.e., it is comprised by more sub-instructions. Since the code that is executed on a processing unit could be interspersed by another, the final value of the “disputed” memory location is unpredictable.

3.1. A brief overview of OpenMP

OpenMP is a portable Application Programming Interface (API) that provides support for shared-memory parallel programming in C/C++ and Fortran for several architectures. The OpenMP specifications were released for the first time by the OpenMP Architecture Review Board (ARB) in October 1997. Among permanent members of the organization there are also market leaders such as Intel, AMD and NEC. Due to the importance of the library in nowadays parallel shared memory applications, this section illustrates a brief overview of use and main directives that are useful for parallelizing MCA based models.

OpenMP is an implementation of a type of parallelism called multithreading, where a master thread “forks” a specified number of other threads and a certain task is divided among them. The threads then run concurrently, with the operating system allocating threads to different processors. When the operating system creates a process (i.e., an instance of a running program), it allocates all resources necessary for the proper execution of the program, including the memory in which to load all necessary data and the program itself. In a multi-threaded program, the code can be executed by multiple threads independently, even though they share the resources of the process which

they are part of (called the *master* or *parent*). Since the execution can take place independently among the various threads of a process, they can be executed simultaneously on different (parallel) processing elements. It is worth to note that, thanks to the fact that threads share most of the resources of the parent process, creating a new thread is less expensive than creating a process. OpenMP uses the multi-threaded fork-join programming model (Fig. 2). In this model, a process is initially constituted by a single thread called initial thread. OpenMP permits to specify some regions of code that can be run in parallel. When a thread encounters one of these regions, it is replicated (fork) many times giving rise to a team of threads that run concurrently. The thread that has given rise to the team is called the master thread of the team.

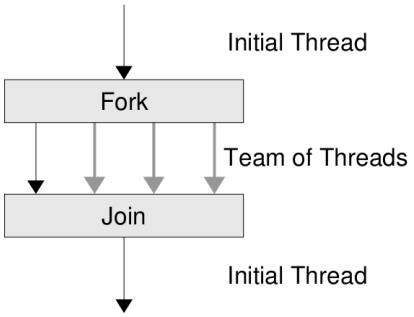

Fig. 2. The OpenMP fork-join shared-memory programming model. An initial master thread replicates a team of threads when a parallel region is encountered. When parallel operations are completed, threads re-join to the master thread (from *Using OpenMP* by *B. Chapman et al.*, *MIT Press,* 2007).

Once that operations in parallel are completed, threads created during the fork phase rejoin to the master thread (join phase). In a typical execution of a parallel program, more stages of fork-join can coexist: this involves a considerable advantage compared with considering the problem in parallel in its entirety [3]. In this way, indeed, parts of code to be parallelized can be selected: one can focus on the parts of code that affect the performance more, while leaving others that, at the cost of considerable difficult parallelization, would lead to derisory improvements or even to a worsening.

Even though using the fork-join model, OpenMP hides low-level implementation details, allowing one to use the same code for more platforms, one still can compile the same source to produce both parallel or sequential (in this case the compiler ignores the OpenMP directives) versions of the program. In addition, within the same program, a function may be performed both in parallel and in sequence, during the same run.

3.2. Basic usage of OpenMP

OpenMP provides the programmer with compiler directives and library functions. These directives allow to define which and how to parallelize certain areas. In particular, they permit to indicate when to create and destroy the team of threads (through the `parallel` directive) and how to redistribute the work among the various threads of the team (cf. *work-sharing* constructs later in text) [3]. The advantage of using a directive is that it does not affect the program coding. In this way, it is extremely straightforward to produce code that can be compiled to create both serial and parallel programs. In particular, OpenMP permits to specify the number of threads that are adopted in a parallel program, which can be defined directly in the code or by means of an environment variable (i.e., `OMP_NUM_THREADS`).

The most important OpenMP construct is the *parallel* one, since it permits to define whether a portion of code should be run in parallel, used in this simple C example program:

```
...
#include <omp.h>
int main (int argc , char_ argv [ ])
{
#pragma omp parallel  // definition of the parallel region
   printf (``Hi there, I'm thread n %d  !\n'', omp_get_thread_num());
```

```
    return 0 ;
}
```

In this case, the initial thread will create the team of threads, each of which will execute the code defined by the directive: the `printf` function. The output of the program run with 4 threads is:

```
Hi there, I'm thread n 3
Hi there, I'm thread n 1
Hi there, I'm thread n 2
Hi there, I'm thread n 0
```

OpenMP also permits *work-sharing* constructs for distributing work (mainly loops) among threads:

- `# pragma omp for` - distributes iterations of a loop among threads;
- `# pragma omp sections` - distributes among threads different portions (regions) of code;
- `# pragma omp single` - Causes the region to be performed by a single thread.

A work-sharing construct will have effect only if met by a team of threads, as in cases where it is contained in a parallel region, otherwise, the execution will be carried out in a sequential manner. Usually, to obtain a data-type parallelization, the `omp for` construct is adopted, while to obtain a functional-type parallelization, the `omp sections` is used. At the end of each work-sharing construct, an implicit barrier is present, meaning that the code execution will continue only when all threads have terminated the work assigned to them within the region.

The constructs seen above are able in most cases to solve most synchronization problems. However, sometimes the programmer needs to have more control and flexibility and a *lock*, a tool for synchronizing various units by exploiting memory variables, may represent a valid and efficient solution. Once a process acquires a lock, no other process can until the process that acquired it releases it. This mechanism is very simple, but at the same time may not be efficient. In addition, if not properly used, its use can lead to deadlock problems, a situation in which a first process A expects the acquisition of a lock previously acquired by a second process B. This latter lock is (directly or indirectly) subordinated to the release of another lock acquired by A. In such a situation A will attend indefinitely for the acquisition of the lock.

In OpenMP, a lock must be firstly initialized by the function `omp_init_lock`. To *acquire* a lock, one must use the `omp_set_lock` function and `omp_unset_lock` to release it. When a thread attempts to acquire a lock that is not available with the `omp_set_lock` function, its execution is suspended until the lock is released (`omp_set_lock` is defined as blocking). A non-blocking alternative consists instead in using the `omp_test_lock` function, which returns zero if the lock cannot be acquired, without blocking the execution of the thread. This is useful for managing the synchronization in an even more flexible way and for using the thread for some other purpose, which should wait for the availability of the lock. A lock-based solution applied for avoiding race conditions in SCIARA will be shown in the following.

Eventually, OpenMP permits some control regarding the scheduling of loop iterations to threads. In the default *static* scheduling, iterations are equally divided, on the basis of a "chunk" parameter, and statically assigned to threads in a round-robin fashion. With the *dynamic* scheduling clause, the iterations are instead assigned to threads on demand. That is, when a specific thread finishes its assigned chunk, it requests another chunk of iterations that are left to be executed, in a typical *master-slave* manner. As reported later in the paper, dynamic scheduling will provide the best results in terms of speed up for the parallelization of SCIARA.

4. Parallelization and performance analysis of SCIARA

The SCIARA Cellular Automata simulation model was implemented by adopting a system of double arrays for the CA space representation, for each substate, one (the *main matrix*) for reading cell neighbor substates and a second (the *support matrix*) for writing the new substate value. This choice has proven to be efficient, since it allows

to separate the substates reading phase from the update phase, after the application of the transition function, ensuring data integrity and consistency in a given step of the simulation.

This implementation offers a great simplicity for parallelization, if:

1. The main matrix is accessed only in reading mode;
2. Update access to the support matrix must be limited to the location of memory associated with the central cell.

If these two rules are respected, the transition function can be executed in parallel on each cell; in fact:

- No race conditions may occur, since access to the main matrix is read-only;
- Each application of the transition function updates a different support matrix memory location.

After applying the transition function to all the cell space, the main matrix must be updated, replacing values with the corresponding support matrix ones. It is worth to note that for a sequential implementation, the second rule can be violated (i.e., updates can occur also for neighbouring cells) offering often benefits both in terms of memory usage and performance. This is especially valid in case of CA models for simulating flows that use flow distribution algorithms based on procedures similar to the Minimization of the Differences [7], such as in SCIARA. In these models, the amount of material that is present in a cell is represented through a *thickness* value. The transition function calculates the value of outflows from the central cell, after which the sum of these values is subtracted from the central cell and properly distributed (i.e., added) to other cells in the CA neighborhood. In this case, therefore, in disagreement with the formal CA definition, the transition function determines a status change for neighboring cells instead of exclusively for the central cell. In any case, this does not represent a problem because it is possible to define a cellular automaton which is perfectly consistent with the formal CA definition.

An alternative approach that would comply with the restrictions of the second rule is to use as many new substates as the number of outflows from the central cell towards neighbouring ones are. This is reflected in the formal definition of SCIARA (cf. [24]). The chosen data structure for the representation of this new substate is an array with the same size as the neighborhood. When all outflows are computed, and therefore all outflow substates are consistent, the actual distribution takes place, producing the new value of the quantity of lava in each cell of the CA. The CA space is scanned again in a second sub-step and each cell reads from a neighbour cell the associated outflow substate with which it corresponds to the quantity of inflowing lava. Besides the obvious overhead caused by the increased memory usage, it is worth to note that this time the same operation that was first performed "on the fly" concurrently with the outflow computation, is now performed in two stages, resulting in an additional overhead in terms of execution time.

A first parallelization of the SCIARA CA model using OpenMP was performed by considering explicit outflow substates for SCIARA and subsequently by simply using pragmas for automatic parallelization of loops scanning the CA matrix, that is, by using the `pragma omp for` work construct (*flow* version). A second parallelization approach consisted in retaining the original serial algorithm (i.e., without the outflow substates), using locks to synchronize parts of code that could lead to race conditions (*lock* version). In this version, to ensure data consistency, a lock was associated to each cell. Each thread that updates values of a certain cell which, in turn, may be updated by another concurrent thread, must first acquire the lock. Obviously, the use of locks involves some extra overhead, since threads cannot concurrently update the same cell. Please note that this is necessary because an update (write) is not an atomic operation, and may cause data incoherency. The use of locks is not only necessary for the updating of the cells in the neighborhood, but also for writing values in the central cell. For illustrative purposes, the following simple C-like code segment shows how lava thickness updates can be achieved properly in this lock version, even in parallel. In this code, `r_Qh` and `rw_Qh` represent the thickness substate (cf. Q_h in the SCIARA definition) main (readable) and support (readable/writable) matrices, respectively.

```
void update_thickness(double **r_Qh, double **rw_Qh)
{
 double out_flow[NEIGHBORS_SIZE];
 int i, j;

 #pragma omp for   // sharing the first loop among active threads
 for (i=0; i<CA_WIDTH;i++)
  for (j=0; j<CA_HEIGHT;j++)
  {
   calculate_out_flows (i, j, out_flows); // calculate outflows vector
```

```
    neigh_array = calculate_neighbors_array (i, j); // calculate neighbor vector
    for (int n=0;n<NEIGHBORS_SIZE;n++)
    {
     // subtract outflows from central cell
     omp_set_lock (locks[i][j]); // lock cell (i,j)
     rw_Qh [i][j]= rw_Qh[i][j] - out_flows[n];
     omp_unset_lock (locks[i][j]); // unlock cell (i,j)

     // add outflows to neighbor cells
     omp_set_lock (locks[neigh_array[i].x][neigh_array[i].y]); // lock neighbour cell
      rw_Qh[neigh_array[n].x][neigh_array[n].y]=
           rw_Qh[neigh_array[n].x][neigh_array[n].y]+out_flows[n];
     omp_unset_lock (locks[neigh_array[i].x][neigh_array[i].y]); // unlock neighbour cell
    }
   }
}

//main
...
#pragma omp parallel  //parallel region definition
while (!simulation_finished)
{
...
update_thickness(r_Qh, rw_Qh);
copy_matrix(r_Qh, rw_Qh);  // swap matrices for next CA step
...
}
```

4.1. Results of performance experiments

Performance and efficiency tests regarding the MCA model SCIARA were carried out on two parallel machines, an 8 core shared memory machine, made up by two Quad Xeon 2.8 GHz (called *Stromboli*) and a HP EliteBook 8540w mobile workstation with an Intel Core2 i7 processor, equipped with 4-core/8-threads 2.8 GHz (called *Miracle2*). Several series of tests were performed on both machines regarding simulations of the 1981, 2001 and 2006 lava flow events (cf. Section 2). Experiments performed on the i7 4-core/8-thread *Miracle2* computer have demonstrated a worse scalability. This could be caused by the processor's Hyper-thread and Turbo Boost technologies. The former permits the adoption of eight "virtual" cores that are considered by the operative system although only 4 physical cores are actually available. The latter, on the other hand, optimizes performances when not all available cores are used by overclocking active cores, causing a "positive" falsification of tests that use few processes. For these reasons, results have to be considered *a fortiori* as preliminary for this machine and are thus omitted in this work. Regarding the Xeon-based machine, the adopted C++ compiler was gcc v. 4.4.5, 64 bit version, using compiler flags: `-O2 -march=native -fopenmp`.

Among other metrics [1] adopted in Parallel Computing for assessing the performance of a parallel program, speed-up measures how much faster a parallel algorithm is compared to the corresponding sequential version. Speed-up is defined as the ratio between the sequential time and the parallel time. Fig. 3 shows wall clock timings (parallel execution time) and speed-up measurements referred to experiments carried out by on the Xeon-based machine and referred only to the 1981 lava flow simulation. Tests performed by taking into account the other two lava flows (i.e., the 2001 and 2006 events) have shown similar scalabilities, even if execution timings were minor due to the considered smaller CA space of these two events, and are thus here omitted. For all considered tests, an optimization was carried out by considering the minimum sub-CA rectangular space containing all active cells in the considered CA step. Since the considered phenomenon is topologically connected (i.e., a simulation starts from few active cells and evolves by activating neighbour cells), this optimization drastically reduces execution times since the sub-rectangle is usually quite smaller than the original CA space.

Firstly, simulations where carried out by considering only this CA optimization (*standard* version), consisting of considering both active cells (i.e., were lava is present) and non-active cells of the sub-CA rectangle mentioned above. In this case, while executions times were obviously the highest, performance in terms of speed-up has proven to be the best of all carried out experiments (see Fig. 3a).

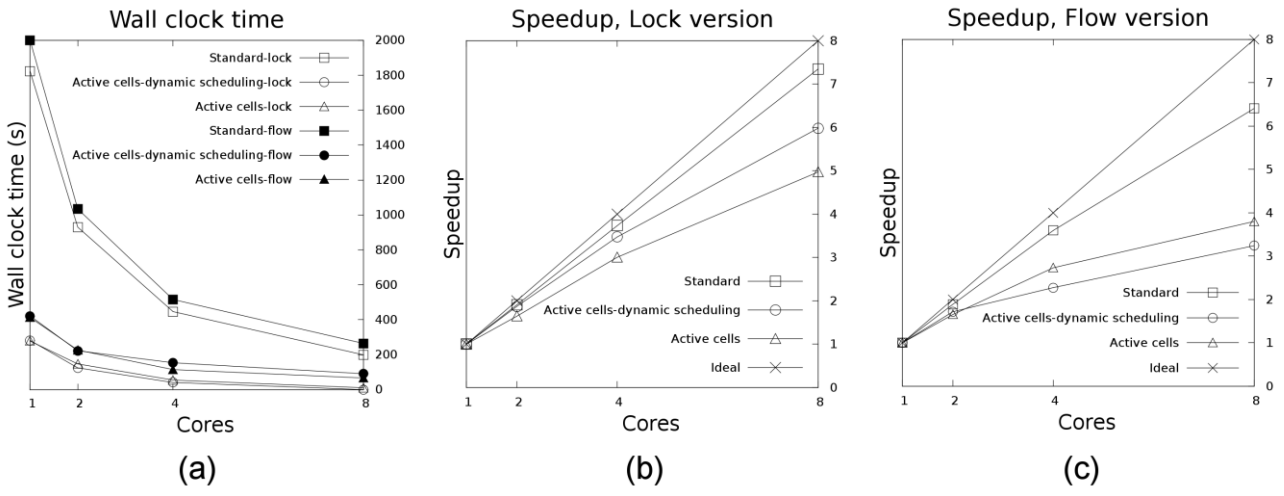

Fig. 3. Performance measurements of experiments carried out on the considered parallel machine, referred to the 1981 event lava flow simulation with the SCIARA model. (a) Wall clock times; (b) Speed-up of the *lock* version; (c) Speed-up of the *flow* version.

This is caused by the fact that the computational load is equi-partitioned among processing elements and thus all cores are more or less equally exploited. Subsequently, a further CA optimization was introduced by considering only active cells (*active cells* version) for the computation of τ, that is, the application of the transition function was not performed for cells that do not contain lava. Speed-up tests carried out considering this latter version were worse than those considering the standard version, even if overall wall clock times for these experiments were smaller. As a matter of fact, for default OpenMP equally partitions loops among available threads, causing an inevitable overload for threads that have more active cells assigned to it. This behaviour can be avoided by exploiting OpenMP's dynamic scheduling (as in the *dynamic-active cells* version), which consists of assigning loop iterations to threads as they request a new loop chunk. When a thread finishes the assigned work, a new loop chunk is assigned to it until the shared loop is terminated, in order to better balance work among the processing elements.

For each of these versions (i.e., *standard*, *active cells* and *dynamic-active cells* versions), the two kinds of parallelizations described in Section 4 (i.e., *lock* and *flow* versions) were carried out, giving rise to six different parallelizations. In these tests, scalability was better for the standard version, even if overall execution times were higher (see Fig. 3a), as reported previously. Still, for all considered tests, the lock version has demonstrated smaller execution timings and a better scalability in terms of speed up than the flow version (Fig. 3b).

In any case, these tests have demonstrated that using more than 8 cores for the considered lava flow model can be useful for future investigations, especially for the *lock* version (Fig. 3b), even if scalability of shared-memory computers is poorer than that of distributed memory machines. In fact, as mentioned previously, in shared memory architectures all processing elements (cores) use the same communication channel (bus) for memory access, representing a bottleneck when frequent memory accesses take place. In addition, with the increase of active processing elements that are considered for experiments, the overhead that is necessary to maintain cache coherence increases drastically. This is certainly the case for the *flow* version (Fig. 3c), which significantly makes more memory accesses than the *lock* version (cf. Section 4). A second reason that prevents a general increase of scalability can be that the considered SCIARA transition function has a low computational cost. While this fact represents certainly a benefit for experiments executed on serial machines, scalability on parallel machines can be a detriment. Similar behaviours have also been verified in other studies, such as fire spread simulation with CA [18].

In conclusion, if absolute execution times have to be considered as crucial in experiments, *all* optimized versions should be taken into account as valid alternatives to the *standard* version which, in turn, even if offering a better scalability, presents greater execution timings. Another motivation of bad performances for the optimized *flow version* of the model might be caused by significant cache thrashing issues [1], the typical problem which is present in multiprocessor architectures when a processor (or core) "finds" its data missing in its cache, due to invalidation by other processors.

5. Conclusions

This work has regarded a first parallelization in OpenMP of the SCIARA CA model for lava flow simulation on shared memory machines. As shown, OpenMP permits a straightforward and portable parallelization, permitting also to maintain the same code for sequential and parallel versions of the program. Although Cellular Automata are considered massively parallel models and standard protocols for their parallelization have been devised, the original (optimized) serial implementation of SCIARA, did not allow to utilize these protocols and therefore it was necessary to adopt different approaches. The adopted strategies were several. Firstly, the sequential version optimizations have been eliminated and the serial code reduced to that of a standard CA by introducing outflow substates, and by applying a standard parallelization. Thereafter, a classical data-parallel version was developed on the basis of the original (optimized) sequential version and explicit synchronization mechanisms (locks) introduced to solve race conditions. The obtained results have shown a general time reduction in experiments thanks to parallelization in all cases, even if a better scalability in terms of speed-up is present in the lock version which, in its dynamic-active cell version, represents the best compromise between execution times and scalability.

The obtained results in this work should be considered more than positive for the parallelization of CA models on shared memory machines, and in view of an extension that allows the execution of the parallel algorithms on hybrid architectures. In fact, the current trend in parallel architectures consists in the adoption of clusters, where each node consists in a multi-core architecture. In this perspective, the goal is to achieve a system that will be able to exploit the parallelism of each node through the shared memory programming paradigm, while taking advantage of more processing nodes through distributed memory programming paradigms. The natural development of this work will therefore be the inclusion of the developed code into a more general coarse-grained system (e.g., based on the message passing paradigm) representing, moreover, one of the first examples of hybrid parallelization of Macroscopic Cellular Automata.

References

1. A. Grama, A. Gupta, G. Karypis, V. Kumar, Introducion to Parallel Computing, 2 Edition. Pearson Education Ltd, Harlow, Essex, 2003.
2. M. Snir, S. Otto, S. Huss-Lederman,D. Walker, J. Dongarra, MPI: The Complete Reference. MIT Press Cambridge, MA, USA, 1995.
3. B. Chapman, G. Jost, R. van der Pas, Using OpenMP Portable Shared Memory Parallel Programming, The MIT Press, 2007.
4. J. von Neumann (Edited and completed by A. Burks), Theory of self-reproducing automata, University of Illinois Press, 1966.
5. S. Succi, The Lattice Boltzmann Equation for Fluid Dynamics and Beyond, Oxford University Press, 2004.
6. A.R. McBirney, T. Murase, Ann. Rev. Ear. Plan. Sci., 12 (1984) 337-357.
7. S. Di Gregorio, R. Serra, Fut. Gener. Comp. Syst., 16 (1999) 259-271.
8. G.M. Crisci, S. Di Gregorio, R. Rongo, W. Spataro, J. Vul. Geo. Res 132 (2004) 253–267.
9. D. D'Ambrosio, W. Spataro, G. Iovine, H. Miyamoto, Env. Model. Soft., 22 (2007) 1417-1436.
10. M.V. Avolio, G.M. Crisci, S. Di Gregorio, R. Rongo, W. Spataro, D. D'Ambrosio, Comput. Geosc., 32 (2006) 897-911.
11. T. Salles, T. Mulde, M. Gaudin, M.C. Cacas, S. Lopez, P. Cirac, Geomorph., 97 (2008) 516-537.
12. G. Folino, G. Mendicino, A. Senatore, G. Spezzano, S. Straface, Paral. Comp. 32 (2006) 357-376.
13. M. Piscitelli, F. Badalamenti, G. D'Anna, S. Di Gregorio, In: Proceedings of EUROSIM 2001 congress (2001)
14. G.A. Trunfio, LNCS 3305 (2004) 725–734.
15. VV.AA. 9th International Conference in Cellular Automata for Research and Industry. ACRI 2010. LNCS 6350, 2010.
16. M. Cannataro, S. Di Gregorio, R. Rongo, W. Spataro, G. Spezzano, D. Talia, Paral. Comput. 21 (1995) 803–824.
17. G. Spingola, D. D'Ambrosio, W. Spataro, R. Rongo, G. Zito, Proceedings of PDPTA (2008) 44-50.
18. E Innocenti, X. Silvani, A. Muzy, D. R.C. Hill., Environ. Model. Soft, 24(7) (2009) 819-831.
19. V. Heuveline, M.J.Krause, J. Latt, Comp Math.Applic., 58 (2009) 1071-1080.
20. G.M Crisci,.S. Di Gregorio, G. Ranieri, Proceedings International AMSE Conference Modelling & Simulation, Paris, France, (1982).
21. D. D'Ambrosio ,W. Spataro, Paral. Comp., 33 (2007) 186-212.
22. S. Park, J.D. Iversen, Geophys. Res. Lett. 11 (1984).
23. M. Dragoni, M. Bonafede, E. Boschi, J. Volcanol. Geotherm. Res., 30(3-4) (1986) 305-325.
24. W. Spataro, M.V. Avolio, V. Lupiano, G.A. Trunfio, R. Rocco, D. D'Ambrosio, Procedia Computer Science, 1, 1, (2010) 17-26.
25. M. J. Flynn, IEEE Trans. Comp., C-21(9) (1972) 948-960.